Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC

José Bacelar Almeida[1,2], Manuel Barbosa[1,3(✉)], Gilles Barthe[4], and François Dupressoir[4(✉)]

[1] HASLab – INESC TEC, Porto, Portugal
[2] University of Minho, Braga, Portugal
jba@di.uminho.pt
[3] DCC-FC, University of Porto, Porto, Portugal
mbb@dcc.fc.up.pt
[4] IMDEA Software Institute, Madrid, Spain
fdupress@gmail.com

Abstract. We provide further evidence that implementing software countermeasures against timing attacks is a non-trivial task and requires domain-specific software development processes: we report an implementation bug in the s2n library, recently released by AWS Labs. This bug (now fixed) allowed bypassing the balancing countermeasures against timing attacks deployed in the implementation of the MAC-then-Encode-then-CBC-Encrypt (MEE-CBC) component, creating a timing side-channel similar to that exploited by Lucky 13.

Although such an attack could only be launched when the MEE-CBC component is used in isolation – Albrecht and Paterson recently confirmed in independent work that s2n's second line of defence, once reinforced, provides adequate mitigation against current adversary capabilities – its existence serves as further evidence to the fact that conventional software validation processes are not effective in the study and validation of security properties. To solve this problem, we define a methodology for proving security of implementations in the presence of timing attackers: first, prove *black-box security* of an algorithmic description of a cryptographic construction; then, establish *functional correctness* of an implementation with respect to the algorithmic description; and finally, prove that the implementation is *leakage secure*.

We present a proof-of-concept application of our methodology to MEE-CBC, bringing together three different formal verification tools to produce an assembly implementation of this construction that is verifiably secure against adversaries with access to some timing leakage. Our methodology subsumes previous work connecting provable security and side-channel analysis at the implementation level, and supports the verification of a much larger case study. Our case study itself provides the first provable security validation of complex timing countermeasures deployed, for example, in OpenSSL.

T. Peyrin (Ed.): FSE 2016, LNCS 9783, pp. 163–184, 2016.
DOI: 10.1007/978-3-662-52993-5_9

1 Introduction

There is an uncomfortable gap between provable security and practical implementations. Provable security gives strong guarantees that a cryptographic construction is secure against efficient *black-box* adversaries. Yet, implementations of provably secure constructions may be vulnerable to practical attacks, due to implementation errors or side-channels. The tension between provable security and cryptographic engineering is illustrated by examples such as the MAC-then-Encode-then-CBC-Encrypt construction (MEE-CBC), which is well-understood from the perspective of provable security [22,26], but whose implementation has been the source of several practical attacks in SSL or TLS implementations. These security breaks are, in the case of MEE-CBC, due to vulnerable implementations providing the adversary with padding oracles, either through error messages [29], or through observable non-functional behaviours such as execution time [2,16]. These examples illustrate two shortcomings of provable security when it comes to dealing with implementations. First, the algorithmic descriptions used in proofs elide many potentially critical details; these details must be filled by implementors, who may not have the specialist knowledge required to make the right decision. Second, attackers targeting real-world platforms may break a system by exploiting side-channel leakage, which is absent in the black-box abstractions in which proofs are obtained.

These shortcomings are addressed independently by *real-world cryptography* and *secure coding methodologies*, both of which have their own limitations. Real-world cryptography [18] is a branch of provable security that incorporates lower-level system features in security notions and proofs (for example, precise error messages or message fragmentation). Real-world cryptography is a valuable tool for analyzing the security of real-world protocols such as TLS or SSH, but is only now starting to address side-channels [8,15] and, until now, has stayed short of considering actual implementations. Secure coding methodologies effectively mitigate side-channel leakage; for instance, the constant-time methodology [13,21] is consensual among practitioners as a means to ensure a *good* level of protection against timing and cache-timing attacks. However, a rigorous justification of such techniques and their application is lacking and they are disconnected from provable security, leaving room for subtle undetected vulnerabilities even in carefully tailored implementations.

In this paper we show how the real-world cryptography approach can be extended – with computer-aided support – to formally capture the guarantees that implementors empirically pursue using secure coding techniques.

1.1 Our Contributions

Recent high-visibility attacks such as Lucky 13 [2] have shown that timing leakage can be exploited in practice to break the security of pervasively used protocols such as TLS, and have led practitioners to pay renewed attention to software countermeasures against timing attacks. Two prominent examples of this are the recent reimplementation of MEE-CBC decryption in OpenSSL [23], which

enforces a constant-time coding policy as mitigation for the Lucky 13 attack, and the *defense in depth* mitigation strategy adopted by Amazon Web Services Labs (AWS Labs) in a new implementation of TLS called s2n, where various fuzzing- and balancing-based timing countermeasures are combined to reduce the amount of information leaked through timing. However, the secure-coding efforts of cryptography practitioners are validated using standard software engineering techniques such as testing and code reviews, which are *not* well-suited to reasoning about non-functional behaviours or cryptography.

As a first contribution and motivation for our work, we provide new evidence of this latent problem by recounting the story of Amazon's recently released s2n library, to which we add a new chapter.

New evidence in s2n. In June 2015, AWS-Labs made public a new open-source implementation of the TLS protocol, called s2n [28] and designed to be "small, fast, with simplicity as a priority". By excluding rarely used options and extensions, the implementation can remain small, with only around 6 K lines of code. Its authors also report extensive validation, including three external security evaluations and penetration tests. The library's source code and documentation are publicly available.[1]

Recently, Albrecht and Paterson [1] presented a detailed analysis of the countermeasures against timing attacks in the original release of s2n, in light of the lessons learned in the aftermath of Lucky 13 [2]. In their study, they found that the implementation of the MEE-CBC component was not properly balanced, and exposed a timing attack vector that was exploitable using Lucky 13-like techniques. Furthermore, they found that the second layer of countermeasures that randomizes error reporting delays was insufficient to remove the attack vector. Intuitively, the granularity of the randomized delays was large enough in comparison to the data-dependent timing variations generated by the MEE-CBC component that they could be 'filtered out' leaving an exploitable side-channel. As a response to these findings, the s2n implementation was patched,[2] and both layers of countermeasures were improved to remove the attack vector.[3]

Unfortunately, this is not the end of the story. In this paper we report an implementation bug in this "fixed" version of the library, as well as a timing attack akin to Lucky 13 that bypasses once more the branch-balancing timing countermeasures deployed in the s2n implementation of MEE-CBC. This implementation bug was subtly hidden in the implementation of the timing countermeasures themselves, which were added as mitigation for the attack reported

[1] https://github.com/awslabs/s2n.

[2] See the details of the applied fixes in https://github.com/awslabs/s2n/commit/4d3729.

[3] We note that the delay randomization countermeasure was further improved since the attacks we describe to sampling the delay between 10 s and 30 s (https://github.com/awslabs/s2n/commit/731e7d). Further, measures were added to prevent careless or rogue application code from forcing s2n to signal decryption failures to the adversary before that delay had passed (https://github.com/awslabs/s2n/commit/f8a155).

by Albrecht and Paterson [1]. We show that the bug rendered the countermeasure code in the MEE-CBC component totally ineffective by presenting a timing attack that breaks the MEE-CBC implementation when no additional timing countermeasures were present. Due to space constraints, details of the attack are given in the full version of the paper.[4]

Disclosure Timeline and Recommendations. The implementation bug and timing attack were reported to AWS Labs on September 4, 2015. The problem was promptly acknowledged and the current head revision of the official s2n repository no longer exhibits the bug and potential attack vector from the MEE-CBC implementation. Subsequent discussions with Albrecht and Paterson and AWS Labs lead us to believe that s2n's second line of defence (the finer grained error reporting delay randomization mechanism validated by Albrecht and Paterson [1]) is currently sufficient to thwart potential exploits of the timing side-channel created by the bug. Therefore, systems relying on unpatched but *complete* versions of the library are safe. On the other hand, any system relying directly on the unpatched MEE-CBC implementation, without the global randomized delay layer, will be vulnerable and should upgrade to the latest version.

The need for formal validation. The sequence of events reported above[5] shows that timing countermeasures are extremely hard to get right and very hard to validate. Our view is that implementors currently designing and deploying countermeasures against side-channel attacks face similar problems to those that were faced by the designers of cryptographic primitives and protocols before the emergence of provable security. On the one hand, we lack a methodology to rigorously characterize and prove the soundness of existing designs such as the ones deployed, e.g., in OpenSSL; on the other hand, we have no way of assessing the soundness of new designs, such as those adopted in s2n, except via empirical validation and trial-and-error. This leads us to the following question: *can we bring the mathematical guarantees of provable security to cryptographic implementations?* We take two steps towards answering this question.

A Case Study: Constant-Time MEE-CBC. Our second and main contribution is the first formal and machine-checked proof of security for an x86 implementation of MEE-CBC in an attack model that includes control-flow and cache-timing channels. In particular, our case study validates the style of countermeasures against timing attacks currently deployed in the OpenSSL implementation of MEE-CBC. We achieve this result by combining three state-of-the-art formal verification tools: i. we rely on EasyCrypt [6,7] to formalize a specification of MEE-CBC and some of the known provable security results for

[4] https://eprint.iacr.org/2015/1241.

[5] The very interesting blog post in http://blogs.aws.amazon.com/security/post/TxLZP6HNAYWBQ6/s2n-and-Lucky-13 analyses these events from the perspective of the AWS Labs development team.

this construction;[6] ii. we use Frama-C to establish a functional equivalence result between EasyCrypt specifications and C implementations; and iii. we apply the CompCert certified compiler [24] and the certified information-flow type-system from [4] to guarantee that the compiled implementation does not leak secret information through the channels considered, and that the compiled x86 code is correct with respect to the EasyCrypt specification proved secure initially.

A Framework for Implementation Security. To tie these verification results together, we introduce — as our third contribution — a framework of definitions and theorems that abstracts the details of the case study. This framework yields a general methodology for proving security properties of low-level implementations in the presence of adversaries that may observe leakage. This methodology relies on separating three different concerns: i. *black-box specification security*, which establishes the computational security of a functional specification (here one can adopt the real-world cryptography approach); ii. *implementation correctness*, which establishes that the considered implementation behaves, as a black-box, exactly like its functional specification; and iii. *leakage security*, which establishes that the leakage due to the execution of the implementation code *in some given leakage model* is independent from its secret inputs. Our main theorem, which is proven using the previous methodology, establishes that our x86 implementation retains the black-box security properties of the MEE-CBC specification, i.e., it is a secure authenticated encryption scheme, even in the presence of a strong timing attacker, and based on standard black-box cryptographic assumptions.

We insist that we do *not* claim to formally or empirically justify the validity of any particular leakage model: for this we rely on the wisdom of practitioners. What we *do* provide is a means to take a well-accepted leakage model, and separately and formally verify, through leakage security, that a concrete deployment of a particular countermeasure in a given implementation does in fact guarantee the absence of any leakage that would weaken a particular security property in the chosen leakage model.

Outline. In Sect. 2, we describe the MEE-CBC construction and informally discuss its security at specification- and implementation-level. We then present the definitions for implementation-level security notions and the statement of our main theorem (Sect. 3). In Sect. 4, we introduce our methodology, before detailing its application to MEE-CBC in Sect. 5. We then present and discuss some benchmarking results in Sect. 6. Finally, we discuss potential extensions to our framework not illustrated by our case study (Sect. 7). We conclude the paper and discuss directions for future work in Sect. 8. A long version of this paper,

[6] Formalizing all known results for MEE-CBC would be beyond the scope of this paper, and we assume that our EasyCrypt specification of the construction inherits all the security properties that have been proved in the literature. In other words, in addition to the properties we formalize, we assume that our MEE-CBC specification satisfies the standard notions of security for authenticated encryption as proved, e.g., by Paterson, Ristenpart and Shrimpton [26].

with appendices including code snippets, formal definitions of standard black-box specification-level security notions, and a discussion of further related work appears on the IACR eprint server.[7]

2 Case Study: MEE-CBC

MAC-then-Encode-then-CBC-Encrypt (MEE-CBC) is an instance of the MAC-then-Encrypt generic construction that combines a block cipher used in CBC mode with some padding and a MAC scheme in order to obtain an authenticated encryption scheme. We consider the specific instantiation of the construction that is currently most widely used within TLS: i. A MAC tag of length tlen is computed over the TLS record header hdr, a sequence number seq and the payload pld. The length of the authenticated string is therefore the length of the payload plus a small and fixed number of bytes. Several MAC schemes can be used to authenticate this message, but we only consider HMAC-SHA256. ii. The CBC-encrypted message m comprises the payload pld concatenated with the MAC tag (the sequence number is not transmitted and the header is transmitted in the clear). iii. The padding added to m comprises plen bytes of value $\mathsf{plen}-1$, where plen may be any value in the range [1..256], such that $\mathsf{plen}+|\mathsf{m}|$ is a multiple of the cipher's block size. iv. We use AES-128 as block cipher, which fixes a 16-byte block size.

At the high level, the HMAC construction computes

$$H((\mathsf{key}_{\mathsf{MAC}} \oplus \mathsf{opad}) \,||\, H((\mathsf{key}_{\mathsf{MAC}} \oplus \mathsf{ipad}) \,||\, \mathsf{hdr} \,||\, \mathsf{seq} \,||\, \mathsf{pld}))\,.$$

We consider a hash function such as SHA-256, which follows the Merkle-Damgård paradigm: a compression function is iterated to gradually combine the already computed hash value with a new 64-byte message block (hash values are tlen bytes long).

Informal security discussion. The theoretical security of MEE-CBC has received a lot of attention in the past, due to its high-profile usage in the SSL/TLS protocol. Although it is well-known that the MAC-then-Encrypt construction does *not* generically yield a secure authenticated encryption scheme [9], the particular instantiation used in TLS has been proven secure [22,25,26]. The most relevant result for this paper is that by Paterson, Ristenpart and Shrimpton [26]. Crucially, their high-level proof explicitly clarifies the need for the implementation to not reveal, in any way, which of the padding or MAC check failed on decryption failures. This is exactly the kind of *padding oracles* exploited in practical attacks against MEE-CBC such as Lucky 13 [2].

After the disclosure of the Lucky 13 attack [2], significant effort was invested into identifying all potential sources of timing leakage in the MEE-CBC decryption algorithm. The implementation subsequently incorporated into OpenSSL, for example, deploys constant-time countermeasures that guarantee the following

[7] https://eprint.iacr.org/2015/1241.

behaviours [23]: i. removing the padding and checking its well-formedness occurs in constant-time; ii. the MAC of the unpadded message is always computed, even for bad padding; iii. the MAC computation involves the same number of calls to the underlying compression function regardless of the number of hash input blocks in the decoded message, and regardless of the length of the final hash block (which may cause an additional block to be computed due to the internal Merkle-Damgård length padding); and iv. the transmitted MAC is compared to the computed MAC in constant-time (the transmitted MAC's location in memory, which may be leaked through the timing of memory accesses, depends on the plaintext length). *Constant-time*, here and in the rest of this paper, is used to mean that the trace of program points and memory addresses accessed during the execution is independent from the initial value of secret inputs. In particular, we note that the OpenSSL MEE-CBC implementation is *not* constant time following this definition: the underlying AES implementation uses look-up table optimizations that make secret-dependent data memory accesses and may open the way to cache-timing attacks.

Our Implementation. The main result of this paper is a security theorem for an x86 assembly implementation of MEE-CBC ($\mathsf{MEE\text{-}CBC}_{\mathrm{x86}}$). The implementation is compiled using CompCert from standard C code that replicates the countermeasures against timing attacks currently implemented in the OpenSSL library [23]. We do not use the OpenSSL code directly because the code style of the library (and in particular its lack of modularity) makes it a difficult target for verification. Furthermore, we wish to fully prove constant-time security, which we have noted is not achieved by OpenSSL. However, a large part of the code we verify is existing code, taken from the NaCl library [14] without change (for AES, SHA256 and CBC mode), or modified to include the necessary countermeasures (HMAC, padding and MEE composition). Our C code is composed of the following modules, explicitly named for later reference: i. $\mathsf{AES128}_{\mathsf{NaCl}}$ contains the NaCl implementation of AES128; ii. $\mathsf{HMACSHA256}_{\mathsf{NaCl}}$ contains a version of the NaCl implementation of HMAC-SHA256 extended with timing countermeasures mimicking those described in [23]; and iii. $\mathsf{MEE\text{-}CBC}_{\mathsf{C}}$ contains an implementation of MEE-CBC using $\mathsf{AES128}_{\mathsf{NaCl}}$ and $\mathsf{HMACSHA256}_{\mathsf{NaCl}}$. We do not include the code in the paper due to space constraints.

As we prove later in the paper, a strict adherence to the coding style adopted in OpenSSL is indeed sufficient to guarantee security against attackers that, in addition to input/output interaction with the MEE-CBC implementation, also obtain full traces of program counter and memory accesses performed by the implementation. However, not all TLS implementations have adopted a strict adherence to constant-time coding policies in the aftermath of the Lucky 13 attack. We now briefly present the case of Amazon's s2n library, discussing their choice of countermeasures, and describing a bug in their implementation that leads to an attack. A more detailed discussion can be found in the long version of this paper.

Breaking the MEE-CBC implementation in s2n. Although parts of the s2n code for MEE-CBC are written in the constant-time style, there are many (intentional) deviations from a strict constant-time coding policy. For example, no attempt is made to de-correlate memory accesses from the padding length value that is recovered from the decrypted (but not yet validated) plaintext. As an alternative, the code includes countermeasures that intend to balance the execution time of secret-dependent conditional branches that might lead to significant variability in the execution time. Roughly, the goal of these countermeasures is to ensure that the total number of calls to the hash compression function is always the same, independently of the actual padding length or validity.

The bug we found resides in a special routine that aims to guarantee that a dummy compression function computation is performed whenever particular padding patterns might lead to shorter execution times. An off-by-one error in the checking of a boundary condition implied that the dummy compression function would be invoked unnecessarily for some padding values (more precisely, there are exactly 4 such padding values, which are easily deduced from the (public) length of the encrypted record).

The leakage the bug produces is similar in size to that exploited by AlFardan and Paterson [2] to recover plaintexts. We have implemented a padding-oracle-style attack on the MEE-CBC decryption routine to recover single plaintext bytes from a ciphertext: one simply measures the decryption time to check if the recovered padding length causes the bug to activate and proceeds by trial and error.[8] The attack can be extended to full plaintext recovery using the same techniques reported in [2].

We already discussed the real-world impact of our attack and our disclosure interaction with AWS Labs in the introduction of this paper. However, we insist that for the purpose of this paper it is *not* the real-world impact of our attack that matters, but the software bug that gave rise to it in the first place. Indeed the existence of such a programming bug and the fact that it remained undetected through AWS Labs' code validation process (and in particular despite unit testing specifically designed to detect timing side-channels) reveal that there is a need for a formal framework in which to rigorously prove that an implementation is secure against timing attacks. This is what we set out to do in the rest of the paper.

3 Security Definitions and Main Theorem

After a brief reminder of the syntax and security notions for secret key encryption relevant to our case study, we introduce and discuss the corresponding implementation-level security notions for the constant-time leakage model and state our main theorem. Cryptographic implementations are often hardwired at a particular security level, which means that asymptotic security notions are not adequate to capture the security guarantees provided by software. We therefore

[8] Plaintext recovery is easier than in Lucky 13, since leakage occurs whether or not the padding string is correct.

omit the security parameter in all our definitions. For simplicity we also keep the running time of algorithms implicit in our notations, although we take care to account for it in our security proofs and to show that there is no hidden slackness in our reductions.

3.1 Secret Key Encryption

We recall that a secret-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is specified as three algorithms: i. a probabilistic key generation algorithm $\mathsf{Gen}(;\mathsf{r})$ that returns a secret key SK on input some random coins r; ii. a probabilistic encryption algorithm $\mathsf{Enc}(\mathsf{m}, \mathsf{SK}; \mathsf{r})$ that returns a ciphertext c on input a message m, the secret key SK, and some random coins r; and iii. a deterministic decryption algorithm $\mathsf{Dec}(\mathsf{c}, \mathsf{SK})$ that returns either a message m or a failure symbol $\perp$ on input a ciphertext c and secret key SK. We denote the set of valid messages with MsgSp and adopt standard notions of correctness, confidentiality ($\mathsf{IND\$\text{-}CPA}$) and integrity ($\mathsf{INT\text{-}PTXT}$ and $\mathsf{INT\text{-}CTXT}$) for authenticated symmetric encryption schemes.

Our goal in the rest of this section is to adapt these standard notions to formally capture implementation-level security. In particular, we wish to give the adversary the ability to observe the leakage produced by the computation of its oracle queries. We first give generic definitions for some core concepts.

3.2 Implementation: Languages, Leakage and Generation

For the sake of generality, our definitions abstract the concrete implementation languages and leakage models adopted in our case study. We later instantiate these definitions with a black-box security model for C implementations and a timing leakage model for x86 assembly implementations.

Language, leakage and machine. Given an implementation language $\mathcal{L}$, we consider a machine $\mathbb{M}$ that animates its semantics. Such a machine takes as input a program P written in $\mathcal{L}$, an input i for P, and some randomness r and outputs both the result o of evaluating P with i and r, and the leakage ℓ produced by the evaluation. We use the following notation for this operation $o \leftarrow \mathbb{M}(P, i; \mathsf{r})_{\rightsquigarrow \ell}$. We make the assumption that the machine is deterministic, so that all randomness required to execute programs is given by the input r. However, our security experiments are probabilistic, and we write $o \leftarrow\!\!\$\, \mathbb{M}(P, i)_{\rightsquigarrow \ell}$ to denote the probabilistic computation that first samples the random coins r that must be passed as randomness input of P, and then runs $\mathbb{M}(P, i; \mathsf{r})$. This approach agrees with the view that the problem of randomness generation is orthogonal to the one of secure implementation [14]. We discuss this further in Sect. 7.

We note that the definition of $\mathbb{M}$ makes three implicit assumptions. First, the semantics of a program must always be defined, since $\mathbb{M}$ always returns a result; termination issues can be resolved easily by aborting computations after a fixed number of steps. Second, our view of $\mathbb{M}$ does not allow an adversary to influence

a program's execution other than through its queries. Finally, our model implies that the semantics of $\mathcal{L}$ can be equipped with meaningful notions of leakage. In the context of our use case, we adopt the common view of practical cryptography that timing leakage can be captured via the code-memory and data-memory accesses performed while executing a program. These can be sensibly formalized over assembly implementations, but not over higher-level implementations (e.g., over C implementations), not least because there is no guarantee that optimizing compilers do not introduce leakage. For this reason, in our case study, we consider the following two implementation models:

- a C-level model using a machine $\mathbb{M}_{\mathrm{C}}^{\emptyset}$ (or simply $\mathbb{M}_{\mathrm{C}}$) that animates the C language semantics with no leakage;
- an assembly-level model using a machine $\mathbb{M}_{\mathrm{x86}}^{\mathsf{CT}}$ that animates (a subset of) the x86 assembly language, and produces leakage traces in the constant-time leakage model as detailed below.

In both languages, we adopt the semantic definitions as formalized in the CompCert certified compiler.

Constant-time leakage traces. Formally, we capture the constant-time leakage model by assuming that each semantic step extends the (initially empty) leakage trace with a pair containing: i. the program point corresponding to the statement being executed; and ii. the (ordered) sequence of memory accesses performed during the execution step. We specify when this particular leakage model is used by annotating the corresponding notion with the symbol CT.

3.3 Authenticated Encryption in the Implementation Model

Given a language $\mathcal{L}$ and a (potentially leaking) machine $\mathbb{M}$ animating its semantics, we now define $\mathbb{M}$-correctness, $\mathbb{M}$-IND$-CPA and $\mathbb{M}$-INT-PTXT security for $\mathcal{L}$-implementations of SKE schemes in the leakage model defined by $\mathbb{M}$. In what follows, we let $\Pi^* = (\mathsf{Gen}^*, \mathsf{Enc}^*, \mathsf{Dec}^*)$ be an SKE implementation in language $\mathcal{L}$.

SKE implementation correctness. We say that Π^* is $\mathbb{M}$-correct if, for all $\mathsf{m} \in \mathsf{MsgSp}$, random coins $\mathsf{r}_{\mathsf{gen}}$, $\mathsf{r}_{\mathsf{enc}}$, and $\mathsf{SK} = \mathbb{M}(\mathsf{Gen}^*; \mathsf{r}_{\mathsf{gen}})$, we have that

$$\mathbb{M}(\mathsf{Dec}^*, \mathbb{M}(\mathsf{Enc}^*, \mathsf{m}, \mathsf{SK}; \mathsf{r}_{\mathsf{enc}}), \mathsf{SK}) = \mathsf{m}\,.$$

SKE implementation security. The $\mathbb{M}$-IND$-CPA *advantage of an adversary* $\mathcal{A}$ *against* Π^* *and public length function* ϕ is defined as the following (concrete) difference

$$\mathbf{Adv}^{\mathbb{M}\text{-ind\$-cpa}}_{\Pi^*,\phi,\mathcal{A}} := \Big|\Pr\Big[\mathbb{M}\text{-IND\$-CPA}^{\mathcal{A}}_{\Pi^*,\phi}(\mathsf{Real}) \Rightarrow \mathsf{true}\Big] - \Pr\Big[\mathbb{M}\text{-IND\$-CPA}^{\mathcal{A}}_{\Pi^*,\phi}(\mathsf{Ideal}) \Rightarrow \mathsf{true}\Big]\Big|\,,$$

where implementation-level game $\mathbb{M}$-IND$-CPA is shown in Fig. 1. Here, public length function ϕ is used to capture the fact that SKEs may partially hide the

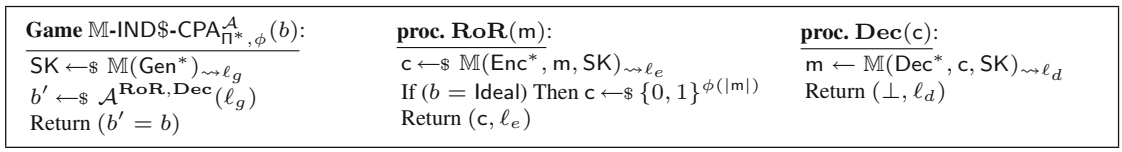

Fig. 1. $\mathbb{M}$-IND$-CPA experiment.

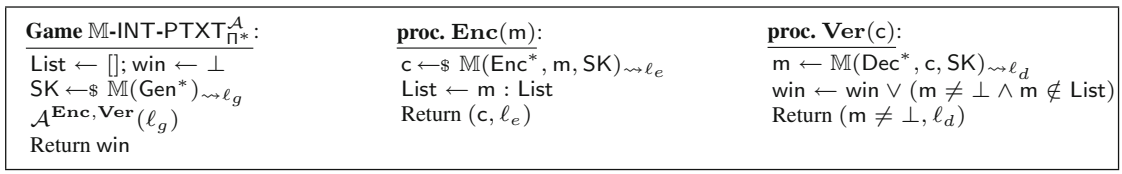

Fig. 2. $\mathbb{M}$-INT-PTXT experiment.

length of a message. If ϕ is the identity function or is efficiently invertible, then the message length is trivially leaked by the ciphertext. In the case of our MEE-CBC specification, for example, the message length is revealed only up to AES block alignment.

We observe that in this refinement of the IND$-CPA security notion for implementations, the adversary may learn information about the secrets via the leakage produced by the decryption oracle Dec^*, even if its functional input-output behaviour reveals nothing. In particular, in a black-box adversary model where leakage traces are always empty, the **Dec** oracle can be perfectly implemented by the procedure that ignores its argument and returns $(\perp, \epsilon)$, and the **RoR** oracle can be simulated without any dependency on m in the Ideal world; this allows us to recover the standard computational security experiment for IND$-CPA. On the other hand, in models where leakage traces are not always empty, the adversary is given the ability to use the decryption oracle with invalid ciphertexts and recover information through its leakage output.

We extend standard INT-PTXT security in a similar way and define the $\mathbb{M}$-INT-PTXT *advantage of an adversary* $\mathcal{A}$ *against* Π^* as the following (concrete) probability:

$$\mathbf{Adv}^{\mathbb{M}\text{-int-ptxt}}_{\Pi^*,\mathcal{A}} := \Pr\left[\mathbb{M}\text{-INT-PTXT}^{\mathcal{A}}_{\Pi^*}() \Rightarrow \mathsf{true}\right],$$

where implementation-level game $\mathbb{M}$-INT-PTXT is shown in Fig. 2.

We similarly "lift" INT-CTXT, PRP (pseudorandomness of a permutation) and UF-CMA (existential MAC unforgeability) security experiments and advantages to implementations. This allows us to state our main theorem.

3.4 Main Theorem

The proof of Theorem 1 is fully machine-checked. However, foregoing machine-checking of the specification's security theorems allows us to strengthen the results we obtain on the final implementations. We discuss this further after we present our proof strategy.

Theorem 1 (CT security of $\mathsf{MEE\text{-}CBC}_{\mathrm{x86}}$). $\mathsf{MEE\text{-}CBC}_{\mathrm{x86}}$ *is* $\mathbb{M}^{\mathsf{CT}}_{\mathrm{x86}}$*-correct and provides* $\mathbb{M}^{\mathsf{CT}}_{\mathrm{x86}}$-$\mathsf{IND\$\text{-}CPA}$ *and* $\mathbb{M}^{\mathsf{CT}}_{\mathrm{x86}}$-$\mathsf{INT\text{-}PTXT}$ *security if the underlying components* $\mathsf{AES128}_{\mathsf{NaCl}}$ *and* $\mathsf{HMACSHA256}_{\mathsf{NaCl}}$ *are black-box secure as a PRP and a MAC, respectively. More precisely, let* $\phi(i) = \lceil (i+1)/16 \rceil + 3$*, then*

- *For any* $\mathbb{M}^{\mathsf{CT}}_{\mathrm{x86}}$-$\mathsf{IND\$\text{-}CPA}$ *adversary* $\mathcal{A}^{\mathsf{cpa}}$ *that makes at most* q *queries to its* **RoR** *oracle, each of length at most* n *octets, there exists an (explicitly constructed)* $\mathbb{M}^{\emptyset}_{\mathsf{C}}$-$\mathsf{IND\$\text{-}CPA}$ *adversary* $\mathcal{B}^{\mathsf{prp}}$ *that makes at most* $q \cdot \lceil (n+1)/16 \rceil + 2$ *queries to its forward oracle and such that*

$$\mathbf{Adv}^{\mathbb{M}^{\mathsf{CT}}_{\mathrm{x86}}\text{-}\mathsf{ind\$\text{-}cpa}}_{\mathsf{MEE\text{-}CBC}_{\mathrm{x86}},\phi,\mathcal{A}^{\mathsf{cpa}}} \leq \mathbf{Adv}^{\mathbb{M}^{\emptyset}_{\mathsf{C}}\text{-}\mathsf{prp}}_{\mathsf{AES128}_{\mathsf{NaCl}},\mathcal{B}^{\mathsf{prp}}} + 2 \cdot \frac{(q \cdot (\lceil \frac{n+1}{16} \rceil + 2))^2}{2^{128}}.$$

- *For any* $\mathbb{M}^{\mathsf{CT}}_{\mathrm{x86}}$-$\mathsf{INT\text{-}PTXT}$ *adversary* $\mathcal{A}^{\mathsf{ptxt}}$ *that makes at most* q_E *queries to its* **Enc** *oracle and* q_V *queries to its* **Ver** *oracle, there exists an (explicitly constructed)* $\mathbb{M}^{\emptyset}_{\mathsf{C}}$-$\mathsf{UF\text{-}CMA}$ *adversary* $\mathcal{B}^{\mathsf{cma}}$ *that makes at most* q_E *queries to its* **Tag** *oracle and* q_V *queries to its* **Ver** *oracle and such that*

$$\mathbf{Adv}^{\mathbb{M}^{\mathsf{CT}}_{\mathrm{x86}}\text{-}\mathsf{int\text{-}ptxt}}_{\mathsf{MEE\text{-}CBC}_{\mathrm{x86}},\mathcal{A}^{\mathsf{ptxt}}} \leq \mathbf{Adv}^{\mathbb{M}^{\emptyset}_{\mathsf{C}}\text{-}\mathsf{uf\text{-}cma}}_{\mathsf{HMACSHA256}_{\mathsf{NaCl}},\mathcal{B}^{\mathsf{cma}}}.$$

In addition, the running time of our constructed adversaries is essentially that of running the original adversary plus the time it takes to emulate the leakage of the x86 implementations using dummy executions in machine $\mathbb{M}_{\mathrm{x86}}$. Under reasonable assumptions on the efficiency of $\mathbb{M}_{\mathrm{x86}}$, this will correspond to an overhead that is linear in the combined inputs provided by an adversary to its oracles (the implementations are proven to run in constant time under the semantics of $\mathcal{L}$ when these inputs are fixed).

Note that the security assumptions we make are on C implementations of AES ($\mathsf{AES128}_{\mathsf{NaCl}}$) and HMAC-SHA256 ($\mathsf{HMACSHA256}_{\mathsf{NaCl}}$). More importantly, they are made in a *black-box* model of security where the adversary gets empty leakage traces.

The proof of Theorem 1 is detailed in Sect. 5 and relies on the general framework we now introduce. Rather than reasoning directly on the semantics of the executable x86 program (and placing our assumptions on objects that may not be amenable to inspection), we choose to prove complex security properties on a clear and simple functional specification, and show that each of the refinement steps on the way to an x86 assembly executable preserves this property, or even augments it in some way.

4 Formal Framework and Connection to PL Techniques

Our formal proof of implementation security follows from a set of conditions on the software development process. We therefore introduce the notion of an implementation generation procedure.

Implementation generation. An implementation generation procedure $\mathcal{C}^{\mathcal{L}_1 \to \mathcal{L}_2}$ is a mapping from specifications in language $\mathcal{L}_1$ to implementations in

Game $\mathsf{Corr}^{\mathcal{A}}_{\mathbb{M},\Pi,\mathcal{C}}()$:	**proc. Eval**(k, i, r):
$\mathsf{bad} \leftarrow \mathsf{false}$	$o \leftarrow \Pi[k](i; \mathsf{r})$
$\Pi^* \leftarrow \mathcal{C}(\Pi)$	$o' \leftarrow \mathbb{M}(\Pi^*[k], i; \mathsf{r})_{\rightsquigarrow \ell}$
$\mathcal{A}^{\mathbf{Eval}}(\Pi^*)$	If $o \neq o'$ then $\mathsf{bad} = \mathsf{true}$
Return $\neg\, \mathsf{bad}$	

Fig. 3. Game defining correct implementation generation. For compactness, we use notation $\Pi[k]$ (resp. $\Pi^*[k]$) for $k \in \{1, 2, 3\}$ to denote the k-th algorithm in scheme Π (resp. implementation Π^*), corresponding to key generation (1), encryption (2) and decryption (3).

language $\mathcal{L}_2$. For example, in our use case, the top-level specification language is the expression language $\mathcal{L}_{\mathrm{EC}}$ of EasyCrypt (a polymorphic and higher-order λ-calculus) and the overall implementation generation procedure $\mathcal{C}^{\mathcal{L}_{\mathrm{EC}} \to \mathcal{L}_{\mathrm{x86}}}$ is performed by a verified manual refinement of the specification into C followed by compilation to x86 assembly using CompCert (here, $\mathcal{L}_{\mathrm{x86}}$ is the subset of x86 assembly supported by CompCert).

We now introduce two key notions for proving our main result: *correct implementation generation* and *leakage security*, which we relate to standard notions in the domain of programming language theory. This enables us to rely on existing formal verification methods and tools to derive intermediate results that are sufficient to prove our main theorem. In our definitions we consider two arbitrary languages $\mathcal{L}_1$ and $\mathcal{L}_2$, a (potentially leaking) machine $\mathbb{M}$ animating the semantics of the latter, and an implementation generation procedure $\mathcal{C}^{\mathcal{L}_1 \to \mathcal{L}_2}$. In this section, $\mathcal{L}_1$ and $\mathcal{L}_2$ are omitted when denoting the implementation generation procedure (simply writing $\mathcal{C}$ instead). In the rest of the paper, we also omit them when clear from context.

Correct implementation generation. Intuitively, the minimum requirement for an implementation generation procedure is that it preserves the input-output functionality of the specification. We capture this in the following definition.

Definition 1 (Correct implementation generation). *The implementation generation procedure $\mathcal{C}$ is correct if, for every adversary $\mathcal{A}$ and primitive specification Π, the game in Fig. 3 always returns* true*.*

For the programming languages we are considering (deterministic, I/O-free languages) this notion of implementation generation correctness is equivalent to the standard language-based notion of simulation, and its specialization as semantic preservation when associated with general-purpose compilers. A notable case of this is CompCert [24] for which this property is formally proven in Coq. Similarly, as we discuss in Sect. 5, a manual refinement process can be turned into a correct implementation generation procedure by requiring a total functional correctness proof. This is sufficient to guarantee *black-box* implementation security. However, it is not sufficient in general to guarantee implementation security in the presence of leakage.

Leakage security. In order to relate the security of implementations to that of black-box specifications, we establish that leakage does not depend on secret inputs. We capture this intuition via the notion of *leakage security*, which imposes that all the leakage produced by the machine $\mathbb{M}$ for an implementation is benign. Interestingly from the point of view of formal verification, leakage security is naturally related to the standard notion of non-interference [19]. In its simplest form, non-interference is formulated by partitioning the memory of a program into *high-security* (or secret) and *low-security* (or public) parts and stating that two executions that start in states that agree on their low-security partitions end in states that agree on their low-security partitions.

We define what the public part of the input means by specifying a function τ that parametrizes our definition of leakage security. For the case of symmetric encryption, for example, τ is defined to tag as public the inputs to the algorithms an attacker has control over through its various oracle interfaces (in IND$-CPA, INT-PTXT and INT-CTXT). More precisely, we define a specific projection function τ_{SKE} as follows:

$$\tau_{\mathsf{SKE}}(\mathsf{Gen}) = \epsilon \qquad \tau_{\mathsf{SKE}}(\mathsf{Enc}, \mathsf{key}, \mathsf{m}) = (|\mathsf{key}|, |\mathsf{m}|) \qquad \tau_{\mathsf{SKE}}(\mathsf{Dec}, \mathsf{key}, \mathsf{c}) = (|\mathsf{key}|, \mathsf{c})$$

Our definition of leakage security then consists in constraining the information-flow into the leakage due to each algorithm, via the following non-interference notion.[9]

Definition 2 ($(\mathbb{M}, \tau)$-non-interference). *Let P be a program in $\mathcal{L}_2$ and τ be a projection function on P's inputs. Then, P is $(\mathbb{M}, \tau)$-non-interferent if, for any two executions $o \leftarrow \mathbb{M}(P, i; \mathsf{r})_{\rightsquigarrow \ell}$ and $o' \leftarrow \mathbb{M}(P, i'; \mathsf{r}')_{\rightsquigarrow \ell'}$, we have $\tau(P, i) = \tau(P, i') \Rightarrow \ell = \ell'$.*

Intuitively, $(\mathbb{M}, \tau)$-non-interference labels the leakage ℓ as a public output (which must be proved independent of secret information), whereas τ is used to specify which inputs of P are considered public. By extension, those inputs that are *not* revealed by τ are considered secret, and are not constrained in any way during either executions. Note that the leakage produced by a $(\mathbb{M}, \tau)$-non-interferent program for some input i can be predicted given only the public information revealed by $\tau(P, i)$: one can simply choose the remaining part of the input arbitrarily, constructing some input i' such that $\tau(P, i) = \tau(P, i')$. In this case, $(\mathbb{M}, \tau)$-non-interference guarantees that the leakage traces produced by $\mathbb{M}$ when executing P on i and i' are equal.

We can now specialize this notion of leakage security to symmetric encryption.

Definition 3 (Leakage-secure implementation generation for SKE). *An implementation generation procedure $\mathcal{C}$ produces $\mathbb{M}$-leakage-secure implementations for SKE if, for all SKE specifications Π written in $\mathcal{L}_1$, we have that the generated $\mathcal{L}_2$ implementation $(\mathsf{Gen}^*, \mathsf{Enc}^*, \mathsf{Dec}^*) = \mathcal{C}(\Pi)$ is $(\mathbb{M}, \tau_{\mathsf{SKE}})$-non-interferent.*

[9] For simplicity, the length of random inputs is assumed to be fixed by the algorithm itself.

Putting the pieces together. The following lemma, shows that applying a correct and leakage secure implementation generation procedure to a black-box secure SKE specification is sufficient to guarantee implementation security.

Theorem 2. *Let $\mathcal{C}$ be correct and produce $\mathbb{M}$-leakage-secure implementations. Then, for all SKE scheme Π that is correct,* IND$-CPA-, INT-PTXT- *and* INT-CTXT-*secure, the implementation $\Pi^* = \mathcal{C}(\Pi)$ is $\mathbb{M}$-correct, $\mathbb{M}$-*IND$-CPA-, $\mathbb{M}$-INT-PTXT- *and* $\mathbb{M}$-INT-CTXT-*secure with the same advantages.*

Proof. Correctness of Π^* follows directly from that of $\mathcal{C}$ and Π. The security proofs are direct reductions. We only detail the proof of $\mathbb{M}$-IND$-CPA, but note that a similar proof can be constructed for $\mathbb{M}$-INT-PTXT and $\mathbb{M}$-INT-CTXT. Given an implementation adversary $\mathcal{A}$, we construct an adversary $\mathcal{B}$ against Π as follows. Adversary $\mathcal{B}$ runs Gen^* on an arbitrary randomness of appropriate size to obtain the leakage ℓ_{Gen} associated with key generation and runs adversary $\mathcal{A}$ on ℓ_{Gen}. Oracle queries made by $\mathcal{A}$ are simulated by using $\mathcal{B}$'s specification oracles to obtain outputs, and the same leakage simulation strategy to present a perfect view of the implementation leakage to $\mathcal{A}$. When $\mathcal{A}$ outputs its guess, $\mathcal{B}$ forwards it as its own guess. We now argue that $\mathcal{B}$'s simulation is perfect. The first part of the argument relies on the correctness of the implementation generation procedure, which guarantees that the values obtained by $\mathcal{B}$ from its oracles in the CPA-game are identically distributed to those that $\mathcal{A}$ would have received in the implementation game. The second part of the argument relies on the fact that leakage-secure implementation generation guarantees that $\mathcal{B}$ knows enough about the (unknown) inputs to the black-box algorithms (the information specified by τ_{SKE}) to predict the exact leakage that such inputs would produce in the implementation model. Observe for example that, in the case of decryption leakage, the adversary $\mathcal{B}$ only needs the input ciphertext c to be able to exactly reproduce the leakage ℓ_{Dec}. Finally, note that the running time of the constructed adversary $\mathcal{B}$ is that of adversary $\mathcal{A}$ where each oracle query $\mathcal{A}$ introduces an overhead of one execution of the implementation in machine $\mathbb{M}$ (which can reasonably be assumed to be close to that of the specification). □

5 Implementation Security of MEE-CBC

We now return to our case study, and explain how to use the methodology from Sect. 4, instantiated with existing verification and compilation tools, to derive assembly-level correctness and security properties for $\mathsf{MEE\text{-}CBC}_{\mathrm{x86}}$.

Proof strategy. We first go briefly over each of the steps in our proof strategy, and then detail each of them in turn in the remainder of this section. In the first step, we specify and verify the correctness and black-box computational security of the MEE-CBC construction using EasyCrypt. In a second step, we use Frama-C to prove the functional correctness of program $\mathsf{MEE\text{-}CBC}_{\mathsf{C}}$ with respect to the EasyCrypt specification. Finally, we focus on the x86 assembly code generated by CompCert ($\mathsf{MEE\text{-}CBC}_{\mathrm{x86}}$), and prove: i. its functional correctness

with respect to the C code (and thus the top-level EasyCrypt specification); and ii. its leakage security. An instantiation of Theorem 2 allows us to conclude the proof of Theorem 1.

Black-box specification security. We use EasyCrypt to prove that the MEE-CBC construction provides IND$-CPA security (when used with freshly and uniformly sampled IVs for each query) and INT-PTXT security.

Lemma 1 (Machine-checked MEE-CBC security). *The following two results hold:*

- *For all legitimate* IND$-CPA *adversary* $\mathcal{A}^{\mathsf{cpa}}$ *that makes at most* q *queries, each of length at most* n *octets, to its* **RoR** *oracle, there exists an explicitly constructed PRP adversary* $\mathcal{B}^{\mathsf{prp}}$ *that makes* $q \cdot \lceil (n+1) \,/\, \lambda \rceil + 2$ *queries to its forward oracle and such that:*

$$\mathbf{Adv}^{\mathsf{ind\$\text{-}cpa}}_{\Pi,\phi,\mathcal{A}} \leq \mathbf{Adv}^{\mathsf{prp}}_{\mathsf{Perm},\mathcal{B}^{\mathsf{prp}}} + 2 \cdot \frac{(q \cdot \lceil \frac{n+1}{\lambda} \rceil + 2)^2}{2^{8 \cdot \lambda}},$$

  where $\phi(i) = \lceil (i+1) \,/\, \lambda \rceil + 3$ *reveals only the number of blocks in the plaintext (and adds to it the fixed number of blocks due to IV and MAC tag).*
- *For all* PTXT *adversary* $\mathcal{A}$ *that makes* q_V *queries to its* **Dec** *oracle, there exists an explicitly constructed* SUF-CMA *adversary* $\mathcal{B}^{\mathsf{cma}}$ *that makes exactly* q_V *queries to its* **Ver** *oracle and such that:*

$$\mathbf{Adv}^{\mathsf{int\text{-}ptxt}}_{\Pi,\mathcal{A}} \leq \mathbf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathsf{Mac},\mathcal{B}^{\mathsf{cma}}}.$$

Our EasyCrypt specification relies on abstract algorithms for the primitives. More precisely, it is parameterized by an abstract, stateless and deterministic block cipher Perm with block size λ octets, and by an abstract, stateless and deterministic MAC scheme Mac producing tags of length $2 \cdot \lambda$.[10] The proofs, formalized in EasyCrypt, are fairly standard and account for all details of padding and message formatting in order to obtain the weak length-hiding property shown in this lemma. Running times for $\mathcal{B}^{\mathsf{prp}}$ and $\mathcal{B}^{\mathsf{cma}}$ are as usual.

We note that, although we have not formalized in EasyCrypt the proof of INT-CTXT security (this would imply a significant increase in interactive theorem proving effort) the known security results for MEE-CBC also apply to this specification and, in particular, it follows from [26] that it also achieves this stronger level of security when the underlying MAC and cipher satisfy slightly stronger security requirements.

Implementation generation. Using Frama-C, a verification platform for C programs,[11] we prove functional equivalence between the EasyCrypt specification and our C implementation. Specifically, we use the deductive verification (WP) plugin to check that our C code fully and faithfully implements a functionality described in the ANSI/ISO C Specification Language (ACSL). To make sure

[10] This is only for convenience in these definitions.
[11] http://frama-c.com/.

that the ACSL specification precisely corresponds to the EasyCrypt specification on which black-box security is formally proved, we rely on Frama-C's ability to link ACSL logical constructs at the C annotation level to specific operators in underlying Why3 theories, which we formally relate to those used in the EasyCrypt proof. This closes the gap between the tools by allowing us to refer to a common specification. Note that, since the abstract block cipher Perm and MAC scheme Mac are concretely instantiated in the C implementation, we instantiate $\lambda = 16$ (the AES block length in bytes) in this common specification and lift the assumptions on Perm and Mac to the C implementation of their chosen instantiation. We then use the CompCert certified compiler [24] to produce our final x86 assembly implementation.

To prove leakage security, we use the certifying information-flow type system for x86 built on top of CompCert [4], marking as public those inputs that correspond to values revealed by τ_{SKE}. Obtaining this proof does not put any additional burden on the user—except for marking program inputs as secret or public. However, the original C code must satisfy a number of restrictions in order to be analyzed using the dataflow analysis from [4]. Our C implementations were constructed to meet these restrictions, and lifting them to permit a wider applicability of our techniques is an important challenge for further work.[12]

Proof of Theorem 1. Let us denote by $\mathcal{C}^{\mathcal{L}_{\mathsf{EC}} \to \mathrm{x86}}$ the implementation generation procedure that consists of hand-crafting a C implementation (annotated with τ_{SKE} consistent security types), equivalence-checking it with an EasyCrypt specification using Frama-C, and then compiling it to assembly using CompCert (accepting only assembly implementations that type-check under the embedded secure information-flow type system), as we have done for our use case. We formalize the guarantees provided by this procedure in the following lemma.

Lemma 2 (Implementation generation). *$\mathcal{C}^{\mathcal{L}_{\mathsf{EC}} \to \mathrm{x86}}$ is a $\mathbb{M}^{\mathsf{CT}}_{\mathrm{x86}}$-correct implementation generation procedure that produces $\mathbb{M}^{\mathsf{CT}}_{\mathrm{x86}}$-leakage secure SKE implementations.*

Proof. Correctness follows from the combination of the Frama-C functional correctness proof and the semantic preservation guarantees provided by CompCert. CompCert's semantics preservation theorem implies that the I/O behaviour of the assembly program exactly matches that of the C program. Functional equivalence checking using Frama-C yields that the C implementation has an I/O behaviour that is consistent with that of the EasyCrypt specification (under the C semantics adopted by Frama-C). Finally, under the reasonable assumption that

[12] In a recent development in this direction, Almeida et al. [3] describe a method, based on limited product programs, for verifying constant-time properties of LLVM code. Their method and the implementation they describe can deal with many examples that the type system from [4] cannot handle, including a less ad hoc version of our code and some of the OpenSSL code for MEE-CBC, whilst preserving a high degree of automation. In addition, their construction easily extends to situations where public outputs are needed to simulate the leakage trace.

the CompCert semantics of C are a sound refinement of those used in Frama-C, we obtain functional correctness of the assembly implementation with respect to the EasyCrypt specification. For leakage security, we rely on the fact that the information-flow type system of [4] enforces τ_{SKE}-non-interference and hence only accepts $(\mathbb{M}^{\mathsf{CT}}_{\text{x86}}, \tau_{\mathsf{SKE}})$-leakage secure implementations. □

Theorem 1 follows immediately from the application of Theorem 2 instantiated with Lemmas 1 and 2. Furthermore, foregoing machine-checking of the black-box specification security proof and simply accepting known results on MEE-TLS-CBC [26], we can also show that $\mathsf{MEE\text{-}CBC}_{\text{x86}}$ is $\mathbb{M}^{\mathsf{CT}}_{\text{x86}}$-INT-CTXT-secure under slightly stronger black-box assumptions on $\mathsf{AES128}_{\mathsf{NaCl}}$ and $\mathsf{HMACSHA256}_{\mathsf{NaCl}}$.

6 Performance Comparison

We now characterize the different assurance/performance trade-offs of the timing mitigation strategies discussed in this paper. Figure 4 shows the time taken by 5 different implementations of MEE-CBC (one of them compiled in different ways) when decrypting a 1.5KB TLS1.2 record using the AES128-SHA256 ciphersuite.[13] More specifically, we consider code from s2n (#1) and OpenSSL (#2), and five different compilations of our formally verified MEE-CBC implementation (#3-7), focusing on raw invocations of MEE-CBC. It is clear that the s2n code (#1) benefits from its less strict timing countermeasures, gaining roughly 1.8× performance over OpenSSL's (semi-)constant-time implementation approach (#2). The figures for our verified implementation of MEE-CBC show both the cost of formal verification and the cost of full constant-time guarantees. Indeed, the least efficient results are obtained when imposing full code and data memory access independence from secret data (#4-6).

#	Implementation	Compiler	Clock Cycles	Time
1	s2n	GCC x86-64 -O2	14K	5μs
2	OpenSSL	GCC x86-64 -O2	23K	9μs
3	$\mathsf{MEE\text{-}CBC}_{\mathsf{C}}$ (AES-NI)	CompCert x86-32	51K	21μs
4	$\mathsf{MEE\text{-}CBC}_{\mathsf{C}}$	GCC x86-64 -O2	59M	25ms
5	$\mathsf{MEE\text{-}CBC}_{\mathsf{C}}$	GCC x86-64 -O1	62M	26ms
6	$\mathsf{MEE\text{-}CBC}_{\text{x86}}$	CompCert x86-32	101M	42ms
7	$\mathsf{MEE\text{-}CBC}_{\mathsf{C}}$	GCC x86-64 -O0	237M	99ms

Fig. 4. Performance comparison of various MEE-CBC implementations. (Median over 500 runs.)

[13] The numbers were obtained in a virtualized Intel x86-64 Linux machine with 4 GB RAM.

The assembly implementation produced using the constant-time version of CompCert (#6), is roughly 8400× slower than s2n, but still over twice as fast as unoptimized GCC. However, the fact that the same C code compiled with `GCC -O2` (#4) is only 1.7× faster[14] than the fully verified CompCert-generated code shows that the bottleneck does not reside in verification, but in the constant-time countermeasures. Indeed, profiling reveals that NaCl's constant-time AES accounts for 97 % of the execution time. These results confirm the observations made in [12] as to the difficulties of reconciling resistance against cache attacks and efficiency in AES implementations. To further illustrate this point, we also include measurements corresponding to a modification of our MEE-CBC implementation that uses hardware-backed AES (#3). This cannot, in fairness, be compared to the other implementations, but it does demonstrate that, with current verification technology, the performance cost of a fully verified constant-time MEE-CBC implementation is not prohibitive.

7 Discussions

On randomness. Restricting our study to deterministic programs with an argument containing random coins does not exclude the analysis of real-world systems. There, randomness is typically scarce and pseudorandom generators are used to expand short raw high-entropy bitstrings into larger random-looking strings that are fed to deterministic algorithms, and it is common to assume that the small original seed comes from an ideal randomness source, as is done in this paper. Our approach could therefore be used to analyze the entire pseudorandom generation implementation, including potential leakage-related vulnerabilities therein.

On length-hiding security. Existing implementations of MEE-TLS-CBC (and indeed our own implementation of MEE-CBC) are not length-hiding as defined in [26] in the presence of leakage. Indeed, the constant-time countermeasures are only applied in the decryption oracle and precise information about plaintext lengths may be leaked during the execution of the encryption oracle. Carrying length-hiding properties down to the level of those implementations may therefore require, either the implementation to be modified (and the Frama-C equivalence proof adapted accordingly), or the specification of implementation security to more closely reflect particular scenarios–such as the TLS record layer–where it may be difficult for the adversary to make chosen-plaintext queries, but easy to make padding and verification oracle queries. In any case, Lemma 1 does capture the length-hiding property given by our choice of minimal padding, and could be adapted to capture the more general length-hiding property of Paterson, Ristenpart and Shrimpton [26] by making padding length a public choice.

[14] This is in line with general CompCert benchmarks.

Leakage simulation and weaker non-interference notions. Our use of leakage security in proving that leakage is not useful to an adversary naturally generalizes to a notion of *leakage simulation*, whereby an implementation is secure as long as its leakage can be efficiently and perfectly simulated from its public I/O behaviour, including its public *outputs*. For example, an implementation of Encrypt-then-MAC that aborts as soon as MAC verification fails, but is otherwise fully constant-time should naturally be considered secure,[15] since the information gained through the leakage traces is less than that gained by observing the output of the **Ver** oracle. The more general notion of leakage simulation informally described here would capture this and can be related to weaker notions of non-interference, where equality on low outputs is only required on traces that agree on the value of public outputs. Theorem 2 can be modified to replace leakage security with the (potentially weaker) leakage simulation hypothesis.

8 Conclusions and Directions for Future Work

Our proposed methodology allows the derivation of strong security guarantees on assembly implementations from more focused and tractable verification tasks. Each of these more specialized tasks additionally carries its own challenges.

Proving security in lower-level leakage models for assembly involves considering architectural details such as memory management, scheduling and data-dependent and stateful leakage sources. Automatically relating source and *existing* assembly implementations requires developing innovative methods for checking (possibly conditional or approximate) equivalences between low-level probabilistic programs. Finally, obtaining formal proofs of computational security and functional correctness in general remain important bottlenecks in the proof process, requiring high expertise and effort. However, combining formal and generic composition principles (such as those used in our case study) with techniques that automate these two tasks for restricted application domains [5,11,20] should enable the formal verification of extensive cryptographic libraries, in the presence of leakage. We believe that this goal is now within reach.

On the cryptographic side, the study of computational security notions that allow the adversary to tamper with the oracle implementation [10] may lead to relaxed functional correctness requirements that may be easier to check, for example by testing. Extensions of our framework to settings where the adversary has the ability to tamper with the execution of the oracle are possible, and would allow it to capture recent formal treatments of countermeasures against fault injection attacks [27].

[15] Some anonymity properties, such as untraceability, may require the cause of decryption failure to remain secret in the black-box model, in which case leakage must not reveal it either [17].

Acknowledgements. The first two authors were funded by Project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/ NORTE-01-0145-FEDER-000020", which is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). The third and fourth authors were supported by projects S2013/ICE-2731 N-GREENS Software-CM and ONR Grants N000141210914 (AutoCrypt) and N00014151 2750 (SynCrypt). The fourth author was supported by FP7 Marie Cure Actions-COFUND 291803 (Amarout II). The machine-checked proof for CBC improves on a script by Benjamin Grégoire and Benedikt Schmidt. Pierre-Yves Strub provided support for extracting Why3 definitions from EasyCrypt specifications. We thank Mathias Pedersen and Bas Spitters for useful comments.

References

1. Albrecht, M.R., Paterson, K.G.: Lucky microseconds: a timing attack on Amazon's s2n implementation of TLS. Cryptology ePrint Archive, report 2015/1129 (2015). http://eprint.iacr.org/
2. AlFardan, N.J., Paterson, K.G.: Lucky thirteen: breaking the TLS and DTLS record protocols. In: IEEE Symposium on Security and Privacy SP 2013, pp. 526–540. IEEE Computer Society (2013)
3. Almeida, J., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. Manuscript (2015). https://fdupress.net/files/ctverif.pdf
4. Barthe, G., Betarte, G., Campo, J.D., Luna, C.D., Pichardie, D.: System-level non-interference for constant-time cryptography. In: Ahn, G.-J., Yung, M., Li, N. (eds.) ACM CCS 2014, pp. 1267–1279. ACM Press, November 2014
5. Barthe, G., Crespo, J.M., Grégoire, B., Kunz, C., Lakhnech, Y., Schmidt, B., Béguelin, S.Z.: Fully automated analysis of padding-based encryption in the computational model. In: Sadeghi, A.-R., Gligor, V.D., Yung, M. (eds), ACM CCS 2013, pp. 1247–1260. ACM Press, November 2013
6. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.-Y.: EasyCrypt: a tutorial. In: Aldini, A., Lopez, J., Martinelli, J. (eds.) FOSAD 2013, pp. 146–166. Springer, Heidelberg (2014)
7. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)
8. Barwell, G., Page, D., Stam, M.: Rogue decryption failures: reconciling AE robustness notions. In: Groth, J., et al. (eds.) IMACC 2015. LNCS, vol. 9496, pp. 94–111. Springer, Heidelberg (2015). doi:10.1007/978-3-319-27239-9_6
9. Bellare, M., Namprempre, C.: Authenticated encryption: relations among notions and analysis of the generic composition paradigm. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 531–545. Springer, Heidelberg (2000)
10. Bellare, M., Paterson, K.G., Rogaway, P.: Security of symmetric encryption against mass surveillance. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 1–19. Springer, Heidelberg (2014)
11. Bernstein, D., Schwabe, P.: Cryptographic software, side channels, and verification. In: COST CryptoAction WG3 Meeting, April 2015
12. Bernstein, D.J.: AES timing variability at a glance (2015). http://cr.yp.to/mac/variability1.html. Accessed 25 Oct 2015

13. Bernstein, D.J.: Cache-timing attacks on AES (2005). Author's webpage
14. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: Hevia, A., Neven, G. (eds.) LatinCrypt 2012. LNCS, vol. 7533, pp. 159–176. Springer, Heidelberg (2012)
15. Boldyreva, A., Degabriele, J.P., Paterson, K.G., Stam, M.: On symmetric encryption with distinguishable decryption failures. In: Moriai, S. (ed.) FSE 2013. LNCS, vol. 8424, pp. 367–390. Springer, Heidelberg (2014)
16. Canvel, B., Hiltgen, A.P., Vaudenay, S., Vuagnoux, M.: Password interception in a SSL/TLS channel. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 583–599. Springer, Heidelberg (2003)
17. Chothia, T., Smirnov, V.: A traceability attack against e-Passports. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 20–34. Springer, Heidelberg (2010)
18. Degabriele, J.-P., Paterson, K.G., Watson, G.J.: Provable security in the real world. IEEE Secur. Priv. **9**(3), 33–41 (2011)
19. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26–28, pp. 11–20. IEEE Computer Society (1982)
20. Hoang, V.T., Katz, J., Malozemoff, A.J.: Automated analysis and synthesis of authenticated encryption schemes. Cryptology ePrint Archive, report 2015/624 (2015). http://eprint.iacr.org/2015/624
21. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 1–17. Springer, Heidelberg (2009)
22. Krawczyk, H.: The order of encryption and authentication for protecting communications (or: how secure is SSL?). In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 310–331. Springer, Heidelberg (2001)
23. Langley, A.: Lucky thirteen attack on TLS CBC. Imperial violet, February 2013. https://www.imperialviolet.org/2013/02/04/luckythirteen.html. Accessed 25 Oct 2015
24. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: ACM Symposium on Principles of Programming Languages POPL 2006 (2006)
25. Maurer, U., Tackmann, B.: On the soundness of Authenticate-then-Encrypt: formalizing the malleability of symmetric encryption. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) ACM CCS 2010, pp. 505–515. ACM Press, October 2010
26. Paterson, K.G., Ristenpart, T., Shrimpton, T.: Tag size *Does* matter: attacks and proofs for the TLS record protocol. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 372–389. Springer, Heidelberg (2011)
27. Rauzy, P., Guilley, S.: A formal proof of countermeasures against fault injection attacks on CRT-RSA. J. Crypt. Eng. **4**(3), 173–185 (2014)
28. Schmidt, S.: Introducing s2n, a new open source TLS implementation, June 2015. https://blogs.aws.amazon.com/security/post/TxCKZM94ST1S6Y/Introducing-s2n-a-New-Open-Source-TLS-Implementation. Accessed 25 Oct 2015
29. Vaudenay, S.: Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 534–546. Springer, Heidelberg (2002)